Functional Programming

Lecture 6: Imperative scheme and parallelism

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

Last lecture

- We do not need to modify the state
- It breaks nice properties of FP
- It can sometimes be useful
  - random access in O(1)
  - data encapsulation

"Classes and objects"

Encapsulate data and related functions

Closures combine data with functions

Assignment allows modifying the data

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT"
                       m))))
  dispatch)
```

Lists modifications

In R5RS, we can modify lists using

`set-car!, set-cdr!`

List are immutable by default with `#lang scheme`

Need to use `mcons, mcar, set-mcar!,…`

Queue

```
(define (make-q)
  (let ((front '()) (back '()))
    (define (in x)
      (let ((new (mcons x '())))
        (if (null? front)
            (begin (set! front new) (set! back new))
            (begin (set-mcdr! back new) (set! back new)))))
    (define (out)
      (let ((x (mcar front)))
        (set! front (mcdr front))
        x))
    (define (dispatch m)
      (cond ((eq? m 'in) in)
                ((eq? m 'out) out)))
    dispatch))
```

Circular lists

```
(define (make-cyclic-list! ls)
  (define (cyc! xs)
    (if (null? (cdr xs))
        (begin (set-cdr! xs ls) ls)
        (cyc! (cdr xs))
        ))
  (cyc! ls))
```

List?

```
(define (my-list? li)
  (define (help? fast slow)
    (cond ((null? fast) #t)
          ((eq? fast slow) #f)
          ((null? (cdr fast)) #t)
          ((eq? (cdr fast) slow) #f)
          ((not (pair? (cdr fast))) #f)
          (#t (help? (cddr fast) (cdr slow)))))
  (cond
     ((null? li) #t)
     ((pair? li) (help? (cdr li) li))
     (#t #f)))
```

Hash table

There are many variants of hash tables

Create a has table comparing with `equal?`

```
(make-hash)
```

Associate `v` with `key` in `hash`

```
(hash-set! hash key v)
(hash-ref hash key [failure-result])
(hash-ref! hash key to-set)
```

`hash-remove!, hash-update!`

Memoization

```
(define (memoize f)
  (let ((table (make-hash)))
    (lambda args
      (hash-ref! table
                 args
                 ;; If the entry isn't there
                 (lambda ()
                    (begin
                      (display "X")
                      (apply f args)) )))))
```

Concurrency and Parallelism

- Thread (concurrency)
  - preempt each other without cooperation
  - share state: variables, function definitions, etc.
  - in Racket, they run on one OS thread
- Futures (parallelism)
  - evaluate an expression in parallel to the main program
  - block on operations that may not run safely in parallel
- Places (parallelism)
  - separate instances of scheme
  - communicate using message passing

Threads

Run on single OS thread

- No speed-up
- Waiting for slow/external event: I/O, sockets, etc.

Operations on threads

- `(thread thunk)` returns thread descriptor
- `thread-suspend, thread-resume, kill-thread`

Channels

- `(make-channel), (channel-put ch v)`
- `(channel-get ch), (channel-try-get ch)`

Thread map

Executes a given function on each element of a list in a separate thread and returns the results

Useful for demonstration purposes

```
(define (thread-map f xs)
  (let ((channels
         (map (lambda(x)
                (let ((c (make-channel)))
                  (thread
                     (thunk (channel-put c (f x))))
                  c
                )
              ) xs)))
    (map channel-get channels)))
```

Futures

`(require racket/future)`

`(future thunk)`

- Starts evaluating an expression (given as thunk)
- Blocks when an operation may not be safely executed
- Returns a "future"

`(touch future)`

- Finish evaluating the expression in the main thread
- If the expression is already evaluated, return the result
- As in *promise*, additional touches just return the result

Future map

Executes a given function on each element of a list in parallel and returns the results

```
(define (future-map f list)
  (let ((res
         (map (lambda (x)
                (future (lambda () (f x))))
              list)))
    (map touch res)))
```

Futures can be visualized and analyzed using

```
(require future-visualizer)
(visualize-futures expr)
```

Home assignment 3

Genetic programming

- Evolution inspired local search in structured data
- Survival of the fittest!!!

- Individual: program for the robot in the maze
- Population: collection of the programs
- New generation: selection, mutation, cross-over
- Fitness function: see Home assignment 2

Summary

- We do not need to modify the state
- It breaks nice properties of FP
- It can sometimes be useful
  - random access in O(1)
  - objects
  - circular data structures
  - memoization
- Concurrency and parallelism